GRASP Version 5.0

Graphical Representations of Algorithms, Structures, and Processes

GRASP/Ada 95
Reverse Engineering Tools For Ada
Final Report
for
Delivery Order No. 33
Basic NASA Contract No. NAS8-39131

Technical Report 96-15
September 29, 1996

Department of Computer Science and Engineering
Auburn University, AL 36849-5347

Contact

James H. Cross II, Ph.D.
Principal Investigator
(334) 844-6315
cross@eng.auburn.edu

GRASP Homepage
http://www.eng.auburn.edu/grasp

NASA
National Aeronautics & Space Administration

Report Documentation Page

1. REPORT NO.	2. GOVERNMENT ACCESSION NO.	3. RECIPIENTS CATALOG NO.

4. TITLE AND SUBTITLE

Reengineering Tools for Use with Ada 95
(GRASP/Ada 95 Tool)

5. REASON DATE

6. PERFROMING ORGANIZATION CODE:
Auburn University
0010090000

7. AUTHORS Dr. James H. Cross II
Principal Investigator

8. PERFORMING ORGANIZATION REPORT NO.
CSE TR 96-15

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Computer Science and Engineering
Auburn Univeristy

10. WORK UNIT NO.
Delivery Order No. 33

11. CONTRACT OR GRANT NO.
NAS8-39131

12. SPONSORING AGENCY NAME AND ADDRESS

NASA/MSFC

13. TYPE OF REPORT AND PERIOD COVERED
Final Report
September 29, 1996
Period Covered:
Apr. 1, 1996 - Sep. 29, 1996

14. SPONSORING AGENCY CODE

15. SUPPLEMENTAL NOTES

NONE

16. ABSTRACT

The GRASP/Ada project (Graphical Representations of Algorithms, Structures, and Processes for Ada) has successfully created and prototyped an algorithmic level graphical representation for Ada software, the Control Structure Diagram (CSD), and a new visualization for a fine-grained complexity metric called the Complexity Profile Graph (CPG). By synchronizing the CSD and the CPG, the CSD view of control structure, nesting, and source code is directly linked to the corresponding visualization of statement level complexity in the CPG. The GRASP v5.0 software tool has been integrated with GNAT, the GNU Ada 95 Translator to provide a comprehensive graphical user interface and development environment for Ada 95. The user may view, edit, print, and compile source code as a CSD with no discernible addition to storage or computational overhead.

The GRASP v5.0 software tool provides the capability for the user to generate CSDs and CPGs from Ada 95 source code in a reverse engineering as well as forward engineering mode with a level of flexibility suitable for practical application. This report provides an overview of the GRASP/Ada project with an emphasis on the current update.

17. KEY WORDS (SUGGESTED BY AUTHORS)	18. DISTRIBUTION STATEMENT
Ada, reengineering, reverse engineering, software engineering, visualization, metrics	Unlimited

19. SECURITY CLASSIFICATION (OF THIS REPORT)	20. SECURITY CLASSIFICATION (OF THIS PAGE)	21. NO. PAGES	22. PRICE
None	None	37	N/A

GRASP Version 5.0

Graphical **R**epresentations of **A**lgorithms, **S**tructures, and **P**rocesses

GRASP/Ada 95
Reverse Engineering Tools For Ada
Final Report
for
Requisition No. 391310005(1F)
Basic NASA Contract No. NAS8-39131

Technical Report 96-11
October 18, 1996

Abstract

The GRASP/Ada project (Graphical Representations of Algorithms, Structures, and Processes for Ada) has successfully created and prototyped an algorithmic level graphical representation for Ada software, the Control Structure Diagram (CSD), and a new visualization for a fine-grained complexity metric called the Complexity Profile Graph (CPG). By synchronizing the CSD and the CPG, the CSD view of control structure, nesting, and source code is directly linked to the corresponding visualization of statement level complexity in the CPG. GRASP has been integrated with GNAT, the GNU Ada 95 Translator to provide a comprehensive graphical user interface and development environment for Ada 95. The user may view, edit, print, and compile source code as a CSD with no discernible addition to storage or computational overhead.

The primary impetus for creation of the CSD was to improve the comprehension efficiency of Ada software and, as a result, improve reliability and reduce costs. The emphasis has been on the automatic generation of the CSD from Ada 95 source code to support reverse engineering and maintenance. The CSD has the potential to replace traditional prettyprinted Ada source code. The current update has focused on the design and implementation of a new Motif compliant user interface, and a new CSD generator consisting of a *tagger* and *renderer*.

The Complexity Profile Graph (CPG) is based on a set of functions that describes the context, content, and the scaling for complexity on a statement by statement basis. When combined graphically, the result is a composite profile of complexity for the program unit. On-going research includes the development and refinement of the associated functions, and the development of the CPG generator prototype.

The current Version 5.0 prototype provides the capability for the user to generate CSDs and CPGs from Ada 95 source code in a reverse engineering as well as forward engineering mode with a level of flexibility suitable for practical application. This report provides an overview of the GRASP/Ada project with an emphasis on the current update.

ACKNOWLEDGEMENTS

The GRASP project has been supported, in part, by grants from NASA, the DoD Advanced Research Projects Agnecy (ARPA), and the Defense Information Systems Agency (DISA).

We appreciate the assistance provided by NASA personnel, especially Mr. Robert Stevens and Judith Gregory. The grants from ARPA and DISA focused on the utilization of GRASP/Ada in Computer Science and Engineering courses at Auburn University and preparation of GRASP/Ada for distribution to other universities.

The following is an alphabetical listing of the team members by category, who have participated in various phases of the project.

Principal Investigator: Dr. James H. Cross II, Associate Professor

Co-Principal Investigator: Dr. Kai H. Chang, Associate Professor

Faculty Investigator: Dr. T. Dean Hendrix, Assistant Professor

Graduate Research Assistants: Larry A. Barowski, Karl S. Mathias, Patricia A. McQuaid, Joseph C. Teate

Undergraduate Research Assistant: Tahia I. Morris

Past Graduate Research Assistants: Richard A. Davis, Charles H. May, Kelly I. Morrison, Timothy A. Plunkett, Brian Randles, Narayana S. Rekapalli, Mark Sadler, Darren Tola

Table of Contents

Page

1. INTRODUCTION 1

2. THE CONTROL STRUCTURE DIAGRAM 4

2.1 The Control Structure Diagram Illustrated 4
2.2 Control Flow Constructs 6
2.3 CSD Box Symbols 7
2.4 CSD Unit Symbols 8
2.5 Data Symbols 10
2.6 Observations 10
2.7 CSD - Future Directions 10

3. GENERATING CONTROL STRUCTURE DIAGRAMS WITH GRASP 11

4. THE COMPLEXITY PROFILE GRAPH 18

4.1 Related Work 18
4.2 The Complexity Profile Graph 19
4.3 CPG - Future Directions 28

5. TOOL VERIFICATION 29

6. SUMMARY AND FUTURE WORK 29

Figures

Page

Figure 1. Code for Binary_Search 4
Figure 2. CSD for Binary_Search 4
Figure 3. Task Controller 5
Figure 4. CSD for Controller 5
Figure 5. CSD Sequence 6
Figure 6. CSD Selection 6
Figure 7. CSD Iteration 6
Figure 8. CSD Box Notation 7
Figure 9. CSD Unit symbols 8
Figure 10. CSD Unit symbols 9
Figure 11. CSD Data Symbols 10
Figure 12. Control Panel 11
Figure 13. File Options 11
Figure 14. Preference Options 11
Figure 15. Window Options 11
Figure 16. Help Options 11
Figure 17. CSD Window 13
Figure 18. CSD File Options 13
Figure 19. CSD Print Options 13
Figure 20. CSD Edit Options 14
Figure 21. CSD Views 14
Figure 22. CSD Templates 14
Figure 23. CSD Window Locator 15
Figure 24. Compiler Options 15
Figure 25. Error Highlighting 15
Figure 26. Run Options 16
Figure 27. CSD Run / Kill 16
Figure 28. CPG Options 16
Figure 29. CSD Help Options 17
Figure 30. Example Ada 95 while loop 20
Figure 31 . BinarySearch CSD in GRASP/Ada 25
Figure 32. BinarySearch CPG in GRASP/Ada 25
Figure 33. Synchronized CSD and CPG in GRASP/Ada 26
Figure 34. CPG showing Content and Total Complexity 26
Figure 35. CPG showing all five metrics plotted separately 27
Figure 36. Complexity Profile Graph for a larger program 28

Tables

Page

Table 1. CPG Segments for a subset of Ada 95 21
Table 3. Token Weights 23
Table 2. Inherent Complexity Weights 23

1. Introduction

Computer professionals have long promoted the idea that graphical representations of software can be extremely useful as comprehension aids when used to supplement textual descriptions and specifications of software, especially for large complex systems. The general goal of the GRASP/Ada research project is the investigation, formulation and generation of *graphical representations of algorithms, structures, and processes for Ada.* This document focuses on the generation or *reverse engineering* of Control Structure Diagrams (CSDs) and Complexity Profile Graphs (CPGs) from Ada 95 PDL or source code for visualization and measurement. The Control Structure Diagram (CSD) is an algorithmic level graphical representation for Ada software. The Complexity Profile Graph (CPG) is a new visualization of a fine-grained complexity metric. By synchronizing the CSD and the CPG, the CSD view of control structure, nesting, and source code is directly linked to the corresponding visualization of statement level complexity in the CPG. GRASP has been integrated with GNAT, the GNU Ada 95 Translator. This has resulted in a comprehensive graphical user interface and development environment for Ada 95. The user may view, edit, print, and compile source code as CSD's with no discernible addition to storage or computational overhead.

The primary impetus for creation and refinement of the CSD is to improve the comprehension efficiency of Ada software and, as a result, improve reliability and reduce costs during design, implementation, testing, and maintenance. The CSD has the potential to replace traditional prettyprinted Ada source code. The recent refinements and extension of the current CSD for Ada 95 include the creation and implementation of architectural-level graphical symbols which will provide a visual link from the CSD to each Ada 95 program unit in the system architecture diagram.

The Complexity Profile Graph (CPG) is based on a set of functions that describes the context, content, and the scaling for complexity on a statement by statement basis. When combined graphically, the result is a composite profile of complexity for the program unit. On-going research includes the development and refinement of the associated functions, and the development of the CPG generator prototype.

Since the overall goal of the GRASP project is to improve the comprehensibility of software, it is important to be able to identify complex areas of source code. The complexity profile graph (CPG) provides the user with the capability to quickly recognize complex areas of source code. The CPG is significant in that it shows the complexity of a program unit as a profile of statement-level complexity metrics rather than a single metric. For example, in the linguistic approach used in Halstead's software science the numbers of distinct and total operators and operands are used to compute the length and volume of a program without regard for program structure or location within the program. The graph theoretic approach used in computing McCabe's cyclomatic complexity yields a metric based on the number of decisions (edges and nodes in the program graph). These traditional metrics are each single numbers used to describe an entire program unit. While there are other metrics that combine the characteristics of software science and cyclomatic complexity, none addresses the overall characteristics of program unit as a visual complexity profile in the way that the CPG does.

With the CSD and CPG synchronized, as the user can scrolls through the CSD, reading and comprehending the source code, the corresponding CPG provides additional complexity information for each statement in the CSD window. Alternatively, as the user scrolls through the CPG to identify areas of high complexity, the CSD is automatically scrolled to display the corresponding source statements.

The GRASP/Ada 95 tool provides the capability for the user to generate CSDs and CPGs from Ada 95 source code with a level of flexibility suitable for experimentation, evaluation, and practical application. It is expected that the new prototype will be integrated with existing CASE tools, in which the primary motivation for the generation of graphical representations is increased support for software life cycle activities, ranging from design through maintenance, with emphasis on visual verification and measurement. These activities should be greatly facilitated by an automatically generated set of "formalized diagrams and graphs" to supplement the source code and other forms of existing documentation. The overall goal of the GRASP/Ada project is to provide the foundation for a CASE (computer-aided software engineering) environment in which reverse engineering and forward engineering (development) are tightly coupled. In such an environment, the user may specify the software in a graphically-oriented language and then automatically generate the corresponding Ada code. Alternatively, the user may specify the software in Ada 95 and then automatically generate the graphical representations either dynamically as the code is entered or as a form of post-processing.

The GRASP/Ada 95 software tool has the potential to be a powerful aid in any environment where Ada 95 is expected to be written and/or read. The tool is particularly suitable for activities during detailed design, implementation, testing, maintenance and reengineering. The CSD is expected to be a valuable aid in comprehension and analysis of overall program structure and flow of control, while the CPG is expected to provide additional valuable insight by providing a visualization of the complexity of both context and content.

DoD and NASA have made a significant investments in Ada 83 and Ada 95 in an effort to improve the quality of software and to control life cycle costs. With the approval of Ada 95 as an ISO standard and the commercial support for Ada 95 compilers and development environments, the promises of Ada are on the verge of becoming widespread reality. However, a major factor in the success of Ada 95 will be availability of state of the art software support tools. Visualization and measurement of complex software systems is an important area of software engineering research. The current GRASP/Ada 95 research attacks both of these problems at a level that can be expected to play a significant role in the overall improvement of the software process with Ada 95. Since much of DoD software development is expected to be affected by Ada 95, this research has the potential for extremely widespread benefits. In particular, the GRASP/Ada 95 methods and tools could be used to reduce life cycle costs by (1) decreasing the time required for new people to comprehend Ada 95 software during original design and implementation, code reviews, and subsequent maintenance, (2) identifying code sections of increased risk, especially in safety-critical applications, and (3) facilitating and encouraging use of Ada 95 with its enhanced support for object-oriented programming, programming in-the-large, and real-time capabilities.

This report focuses on the Ada 95 aspects of the GRASP environment. However, GRASP has become a very robust software development application and now provides CSD support for C and Java, in addition to Ada 95. Since GRASP is in a continual state of enhancement, readers are referred to the GRASP Homepage (http://www.eng.auburn.edu/grasp) which includes sections on

the CSD, Current Features, FTP Information, an Upgrade Table, Documentation, Future Plans, and Contact Information.

As an aid to those unfamiliar with GRASP and the CSD, we have made documentation available on-line. This information contains an introduction to the CSD, a preliminary on-line user's guide, as well as links to other on-line articles relating to GRASP and the CSD.

Finally, if you have any questions not answered in this document, bugs to report, or general comments to make about GRASP, please contact the author at the email address on the cover of this report.

2. The Control Structure Diagram

The CSD is designed to provide the user with the combined advantages of a graphical notation and prettyprinted source code. Whereas graphical representations such as the flowchart, Warnier-Orr diagram [Orr77], and Nassi-Shneiderman chart [Nassi73] disrupt the familiar flow of well-indented source code, the CSD seeks to preserve this. The philosophy is to increase comprehensibility by augmenting the source code with a graphical notation rather than presenting it to the user in a new graphical layout. The action diagram [Martin85] provides an in line bracket notation, but does not present the "connected" flow of control nor the rich symbols provided by the CSD. Tripp cites many additional graphical representations for programs that have been put forth but are not widely known [Tripp89]. However, none of these has successfully combined the attributes of simplicity, intuitiveness, ease of use, and conciseness as the CSD has. The CSD notation also motivated the creation of a CSD editor which has the look and feel of a typical modern text editor. Again, this allows the user to work in a familiar setting but with the added value of an automatically generated graphical notation. In this section the CSD is introduced, and in Section 3 the automated support provided by the GRASP CSD Window is described.

2.1 The Control Structure Diagram Illustrated

Two examples are presented below to illustrate the CSD. The first shows the basic control constructs of sequence, selection and iteration in Ada. These three control constructs are common to all structured procedural languages such as Ada, C, and Pascal. The second example illustrates a more complex control construct, the task rendezvous in Ada.

Figure 1 contains an Ada function called Binary_Search that searches an array A of elements and counts the number of elements above, below, and/or equal to a specified element. Figure 2 contains the CSD for Binary_Search which includes the three basic control constructs

```
function Binary_Search
      (Key : in KeyType;
       A   : in ArrayType)
   return integer is

      low, middle, high : integer;

   begin
      low  := A First;
      high := A Last;
      while (low <= high) loop
         middle := (low + high) / 2;
         if (Key < A(middle)) then
            high := middle - 1;
         elsif (Key > A(middle)) then
            low := middle + 1;
         else
            return middle;
         end if;
      end loop;
      return 0;
   end Binary_Search;
```

Figure 1. Code for Binary_Search

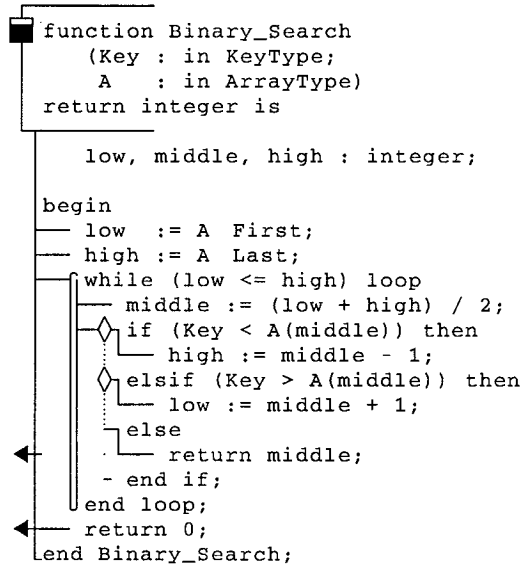

Figure 2. CSD for Binary_Search

sequence, selection, and iteration. Although this is a very simple example, the CSD clearly indicates the levels of control inherent in the nesting of control statements. For example, at level 1 there are four statements executed in sequence - the two assignment statements, a `while` loop, and a `return`.. The `while` loop defines a second level of control which contains a single assignment statement and an `if` statement, which in turn defines three separate level 3 sequences, each of which contains one statement, the last of which is a `return` statement. It is noteworthy that even the CSDs for most production strength procedures generally contain no more than ten to fifteen statements at level 1 or in any of the subsequences defined by control constructs for selection and iteration. This graphical chunking on the basis of functionality and level of control appears to have a substantial positive effect on detailed comprehension of the software.

Figure 3 and Figure 4 contain an Ada task body Controller adapted from [BAR84], which loops through a priority list attempting to accept selectively a Request with priority P. Upon acceptance, some action is taken, followed by an exit from the priority list loop to restart the loop with the first priority. In typical Ada task fashion, the priority list loop is contained in an outer infinite loop. This short example contains two threads of control: the rendezvous, which enters and exists at the accept statement, and the thread within the task body. In addition, the priority list loop contains two exits: the normal exit at the beginning of the loop when the priority list has been exhausted, and an explicit exit invoked within the select statement. While the concurrency and multiple exits are useful in modeling the solution, they do increase the effort required of the reader to comprehend the code.

The CSD in Figure 4 uses intuitive graphical constructs to depict the point of rendezvous, the two nested loops, the select statement guarding the accept statement for the task, the unconditional exit from the inner loop, and the overall control flow of the task. When reading the code without the diagram, as shown in Figure 3, the control constructs and control paths are much less visible although the same structural and control information is available. With additional levels of nesting and increased physical separation of sequential components, the visibility of control constructs and control paths becomes increasingly obscure, and the effort required of the reader

```
task body Controller is

begin
   loop
      for P in Priority loop
         select

            accept Request(P) do

               Action(D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end Controller;
```

Figure 3. Task Controller

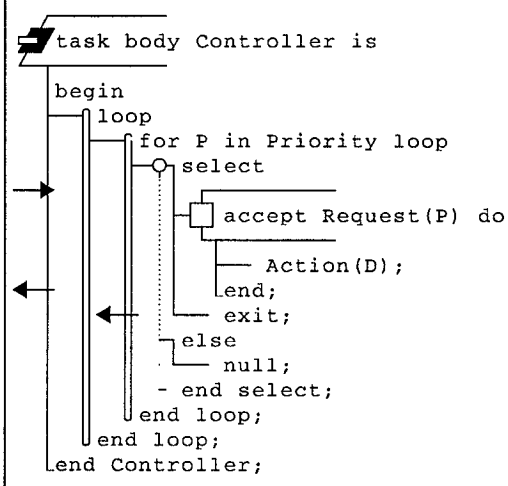

Figure 4. CSD for Controller

dramatically increases in the absence of the CSD.

2.2 Control Flow Constructs

A complete set of CSD graphical constructs has been developed which includes each of the control structures in Ada 95. The basic constructs for sequence, selection, and iteration are illustrated below. Sequence, shown in Figure 5, is represented by a solid vertical line with stems marking the beginning of each statement in the sequence.

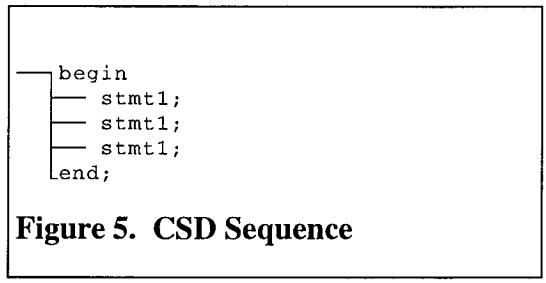

Figure 5. CSD Sequence

The basic constructs for selection are illustrated in **Figure 6**. The familiar diamond is used to indicate a decision and is placed to the left of the condition. The true path is shown with a solid line and the false path is indicated by a dashed line. This becomes very important when selection constructs are nested. The next statement to be executed is found by scanning to the left, skipping through one or more dashed lines, to find the solid vertical line and the next stem. The basic constructs for iteration are shown in Figure 7. Since the CSD supplements the code, the meaning of the CSD is self-evident.

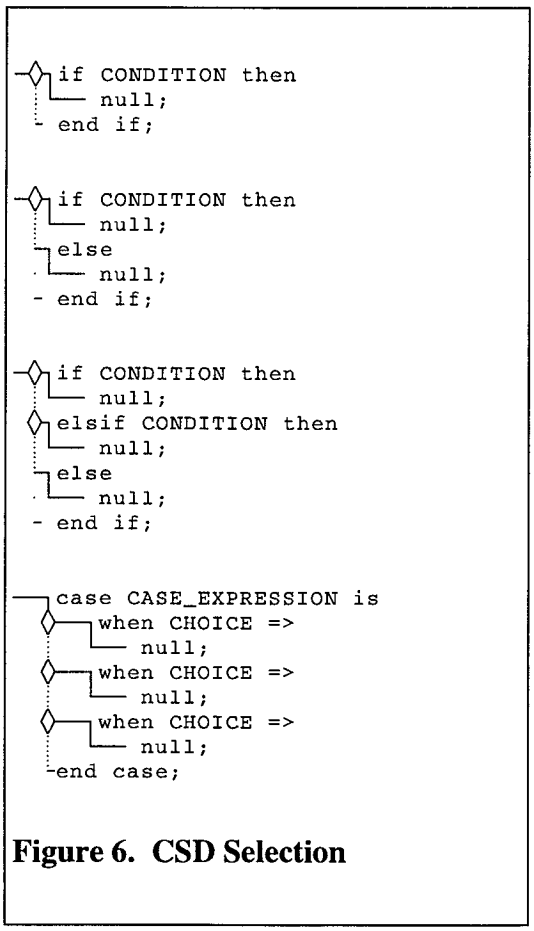

Figure 6. CSD Selection

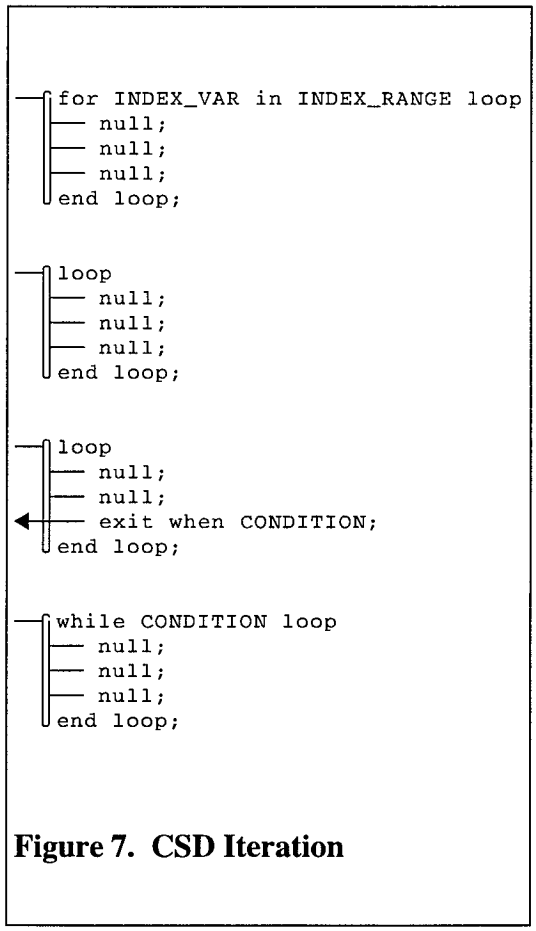

Figure 7. CSD Iteration

2.3 CSD Box Symbols

The CSD uses five different types of open-ended boxes to identify the major Ada program units. These are called the single box, the double box, and the slanted box, and the single and double boxes with dashed lines. Each particular box represents a specific group of Ada program units. Collectively these are referred to as the CSD Box Notation as illustrated in Figure 8.

The single box, which is shown encasing "`procedure Binary_Search is`" in Figure 2, is used to identify both specifications and bodies for Ada subprograms (functions and procedures), protected types, exception handlers, and task entries. The double box is used to identify package and protected specifications and bodies in Ada. The slanted box is used to identify the body and specification of the Ada unit known as the task. Finally, the CSD identifies generic packages and subprograms with a dashed single box for generic subprograms and a dashed double box for generic packages.

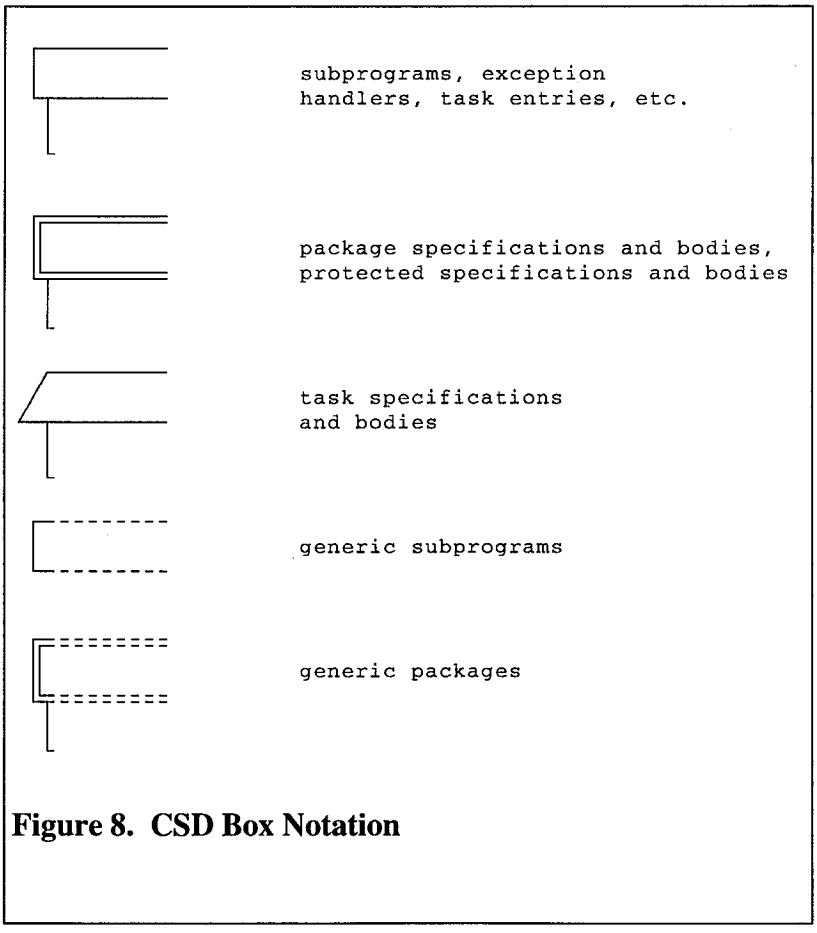

Figure 8. CSD Box Notation

2.4 CSD Unit Symbols

CSD unit symbols, illustrated in Figure 9, provide the user with the option of specializing the program unit identified by the box notation. These are patterned after Booch's module notation [Booch94] but include additional original symbols for *task entry*, *protected specification* and *body*, and *exception handler*. The CSD shown in Figure 2 uses the subprogram unit symbol combined with the box notation, but it could have been just as easily shown using only the box symbols. As programs increase in size and complexity, the CSD unit symbols become more useful in comprehending the Ada source since they can provide a direct visual connection with the architectural diagrams of the system. Many users have indicated they prefer to combine the box notation and the unit symbols as shown in Figure 10.

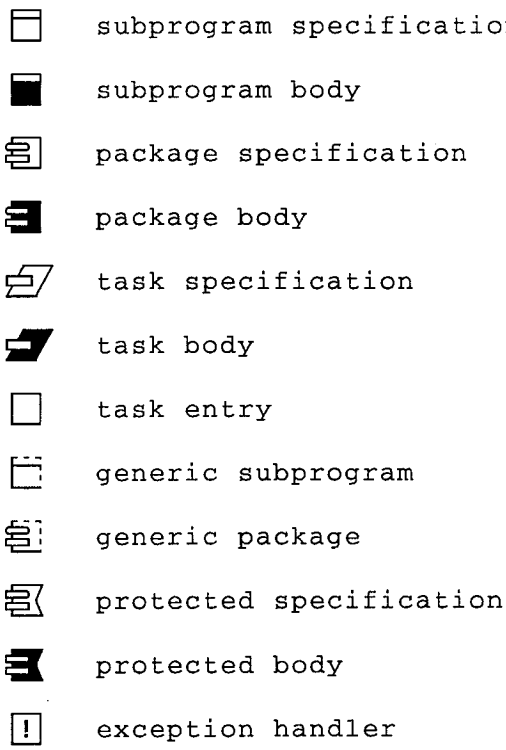

Figure 9. CSD Unit symbols

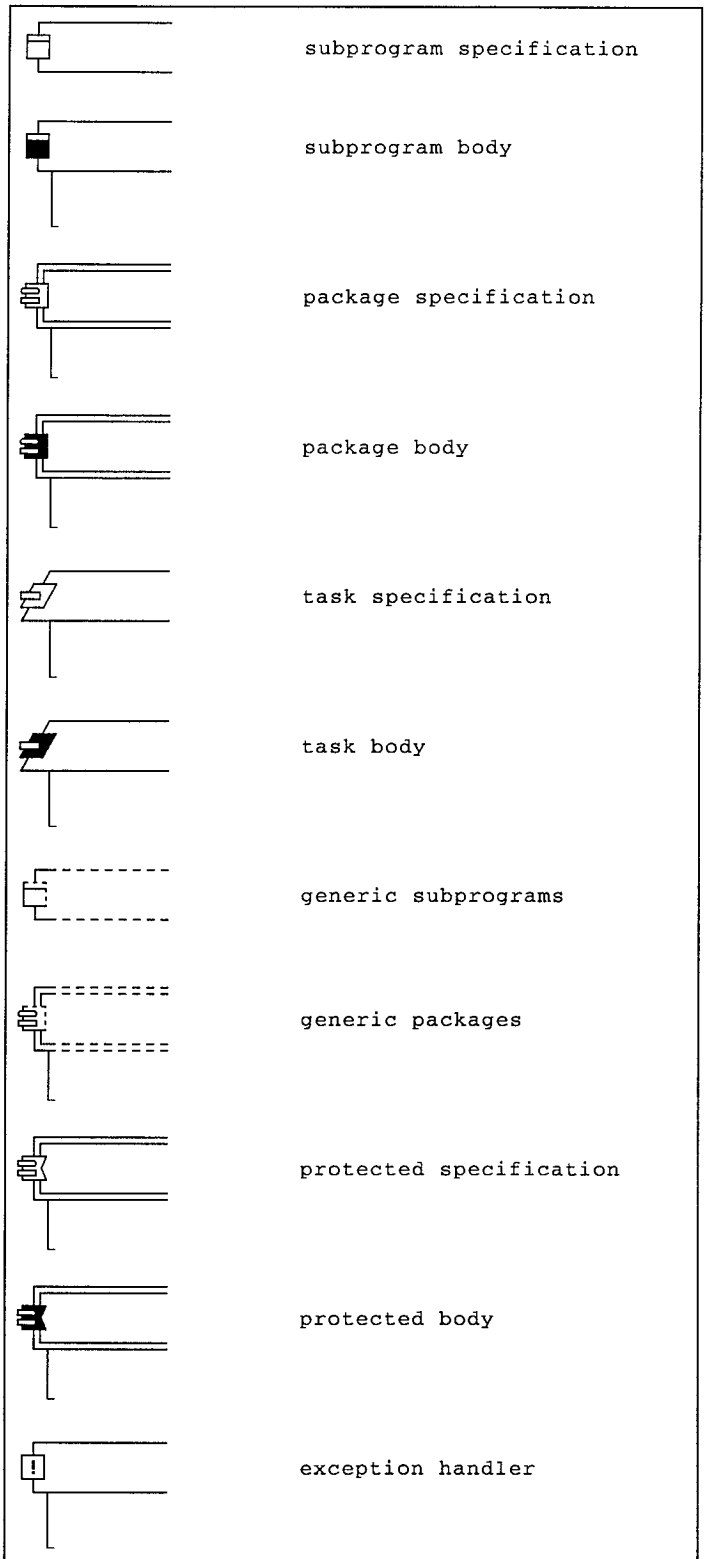

Figure 10. CSD Unit symbols

2.5 Data Symbols

Although the CSD is primarily intended to depict control structures and control flow, many users have found it beneficial to also have distinguishing symbols for type declaration and variable declaration as shown in Figure 11. In the GRASP CSD Window described in Section 3, the user may turn this option, as well as several others, on or off.

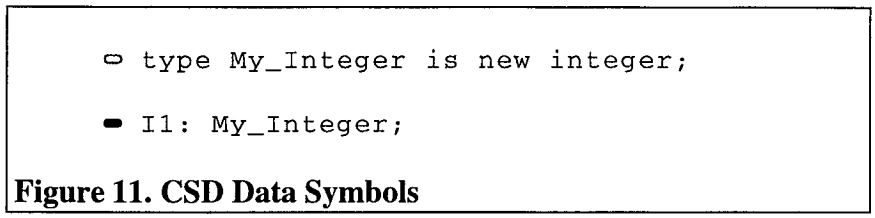

Figure 11. CSD Data Symbols

2.6 Observations

The control structure diagram is a graphical notation which maps directly to Ada 95 and other languages such as C/C++ and Java. The CSD offers advantages over previously available diagrams in that it combines the best features of well-indented code with simple intuitive graphical constructs. The potential of the CSD can be best realized during detailed design, implementation, verification and maintenance. The CSD can be used as a natural extension to popular architectural level representations such as data flow diagrams, object diagrams, and structure charts.

2.7 CSD - Future Directions

The CSD constructs shown in figures above are expected to continue to evolve, especially as the CSD is adapted to additional languages such as C/C++ and Java. The GASP/Ada 95 prototype, described in Section 3, provides for the automatic generation of the CSD for Ada 95, C, and Java source code. Suggestions for improvements to the individual CSD graphical constructs are continually solicited from users.

3. Generating Control Structure Diagrams with GRASP

GRASP is a software engineering tool that generates the CSD for a given Ada program unit, and provides a seamless integration with GNAT to perform other functions associated with code development. GRASP is used to create, edit, compile, and run Ada programs. This section introduces the GRASP environment and provides a brief overview of its most common features.

The GRASP/Ada 95 Version 5.0 prototype, provides a **Control Panel**, shown in Figure 12, for the overall coordination of the environment. From the Control Panel, the user can open one or more CSD windows for Ada 95, C, and Java (Figure 13), set general preferences (Figure 14), locate the GRASP Message Window (Figure 15), or get help on each of the options on the Control Panel or GRASP in general (Figure 16).

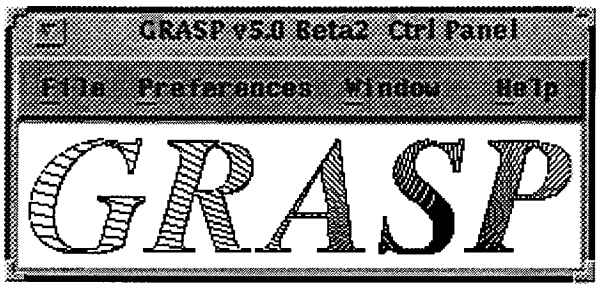

Figure 12. Control Panel

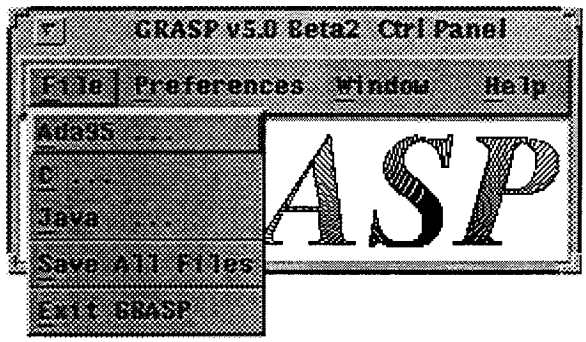

Figure 13. File Options

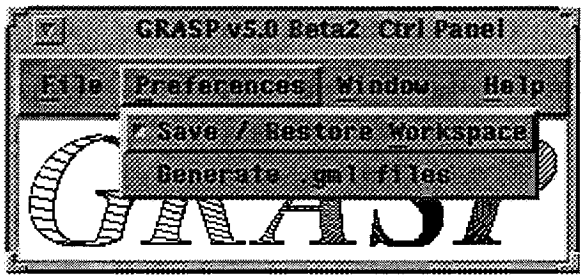

Figure 14. Preference Options

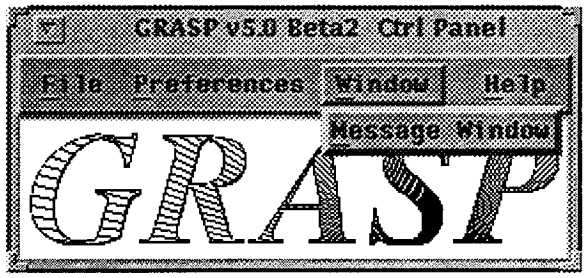

Figure 15. Window Options

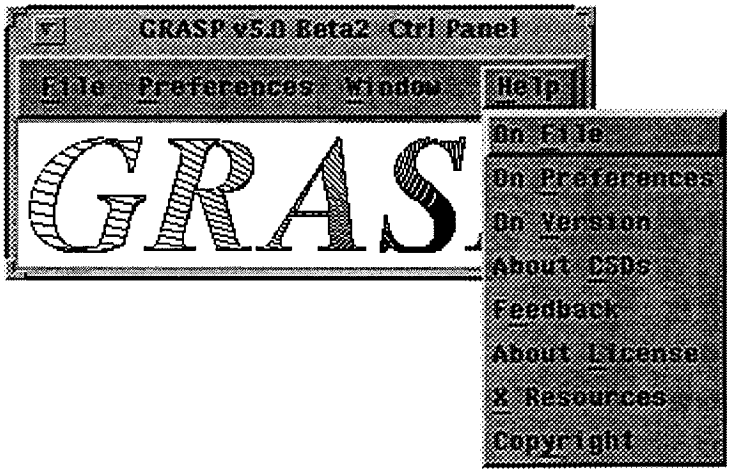

Figure 16. Help Options

The **CSD window**, shown in Figure 17, is a full-function text editor with the additional capability to generate, display, edit, and print CSDs. When a file containing an Ada program unit is loaded, the CSD is automatically generated if **Auto** (next to **Generate CSD** on tool bar) is turned on. Otherwise, the user may generate the CSD on demand by clicking the **Generate CSD** button (or ctrl-g or F1), which is usually done routinely during the course of editing to redraw the diagram. All white space and comments in the source code are preserved with the exception of indentation, which is replaced by the CSD. If a parse error is encountered during CSD generation, the cursor is moved to the highlighted line containing the error to aid the user in making corrections. When the user saves a file, the CSD is filtered so that only the Ada source code is retained. The CSD generation and display cycle is extremely fast (approx. 5,000 lines/sec on a Sparc 10). The net result is that the CSD window can be used in place of a traditional program editor to generate, display, edit, and print CSDs with virtually no overhead; i.e., the CSD is essentially free.

The **File** options, Figure 18, are similar to traditional text editors. The **Print (PostScript)** option, Figure 19 allows the user to set page margins, font size, headings, and number of columns. It also informs the user if there will be line wrap at the current settings. **Default Settings** allows the user to save the current settings of the CSD Window or load the previously saved settings.

Edit, in Figure 20, also includes a block comment and uncomment option. **View**, in Figure 21, allows the user to select any combination (or none) of the following: the standard CSD Box notation, program unit symbols, data symbols, intra-statement alignment, and line numbers. The **Template** option, in Figure 22, opens a tear-off menu of selectable templates for the language of the CSD window (e.g., Ada95, C, or Java). When a template name is clicked, the source code for it is inserted at the point of the cursor. The CSD window Locator in Figure 23 allows the user to quickly locate the Control Panel, Message Window, and Run Shell that go with that particular CSD Window. This is an important feature if more than one copy of GRASP is running.

Version 5.0 is coupled with the GNAT Ada 95 compiler [ACT96]. The CSD window in Figure 24 allows the user to invoke GNAT directly for the current program unit to perform a **Make**, **Compile**, or **Semantic Check**. When an error is reported by the compiler, the offending line of code is highlighted in the diagram. In Figure 25, line 12 is highlighted to indicate that `Counter` has not been defined. Note the error message returned by GNAT is displayed in the GRASP Message window also shown in Figure 25.

After making an executable, the user may run the file directly from the CSD Window by selecting **Run**, **Run Previous**, or **Run File** as shown in Figure 26. **Run** assumes the user wants to run the executable file associated with the source file in the current CSD window. **Run Previous** runs the file that was executed by the most recent of the Run options. **Run File** opens up a file select dialog box, and allows the user to run an existing executable. The **Grasp Run Shell Window** is opened for input/output to the executing program as shown in Figure 27. This shell runs as a separate process so that the execution of the user's program cannot affect GRASP. A **Grasp Run Control** dialog box, also shown in Figure 27, allows the user to send various signals to the program (e.g., interrupt or kill).

The **CPG** options shown in Figure 28 are described in Section 4. Currently, this option is only available in the Ada 95 CSD Window. Finally, the **Help** option, shown in Figure 29, provides a detailed description of each feature in the CSD Window.

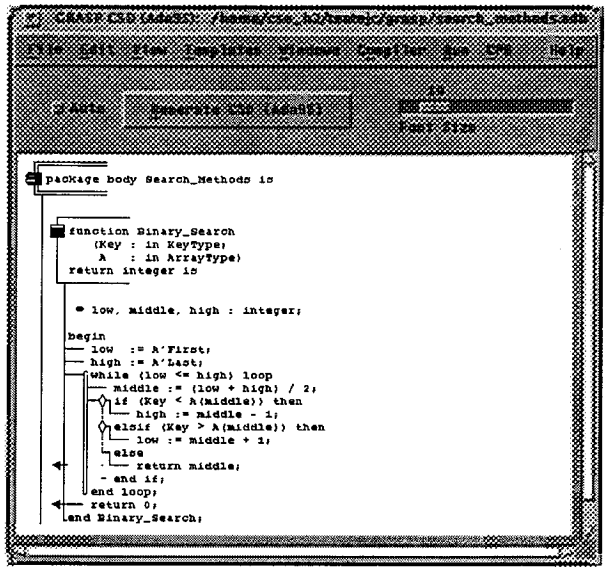

Figure 17. CSD Window

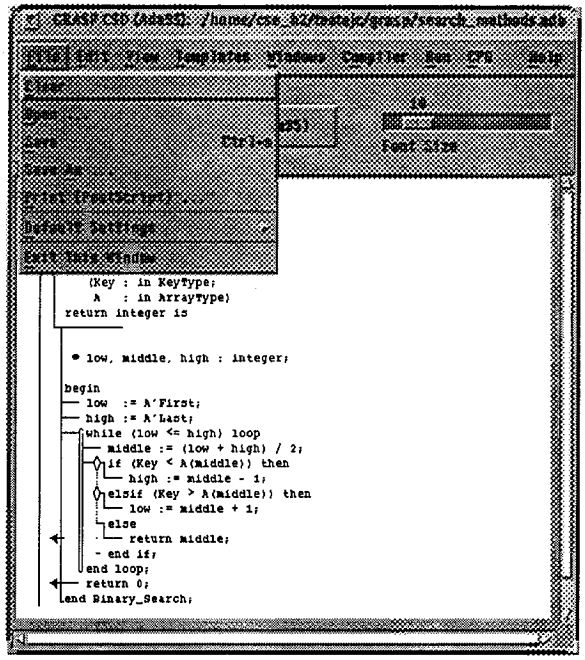

Figure 18. CSD File Options

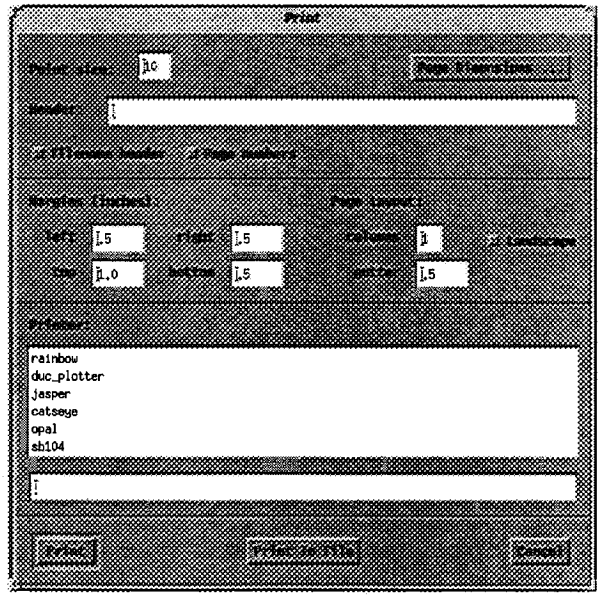

Figure 19. CSD Print Options

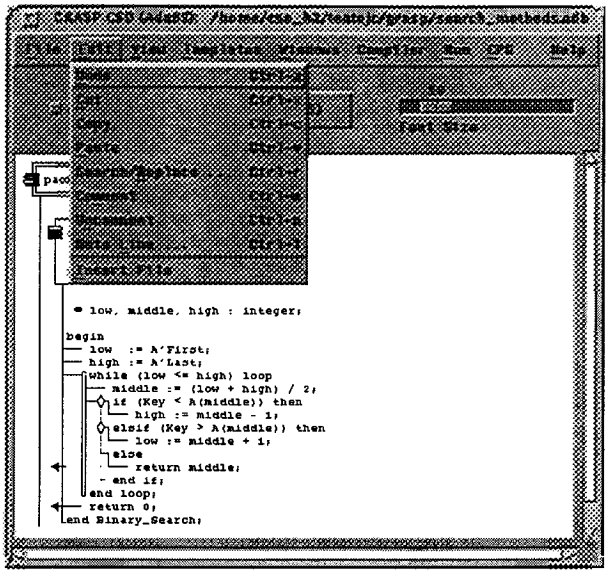

Figure 20. CSD Edit Options

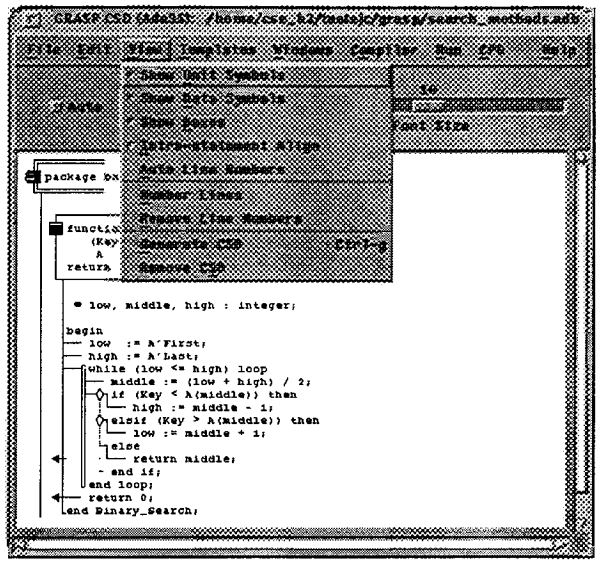

Figure 21. CSD Views

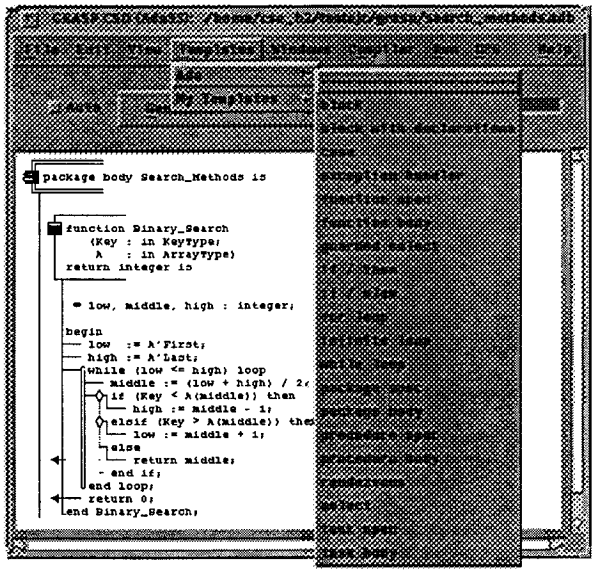

Figure 22. CSD Templates

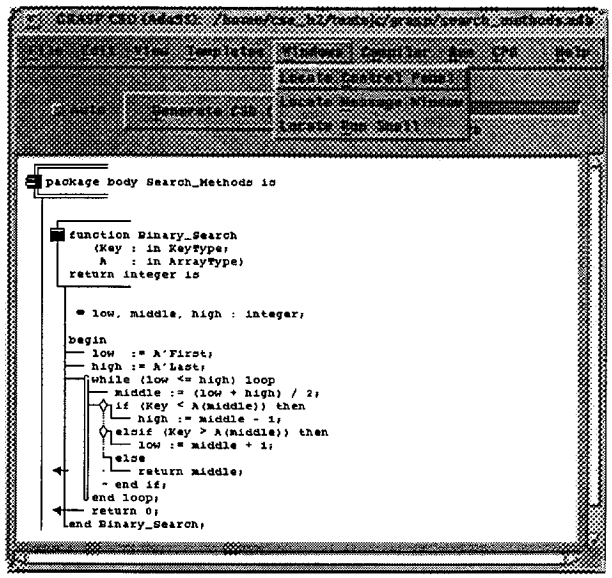

Figure 23. CSD Window Locator

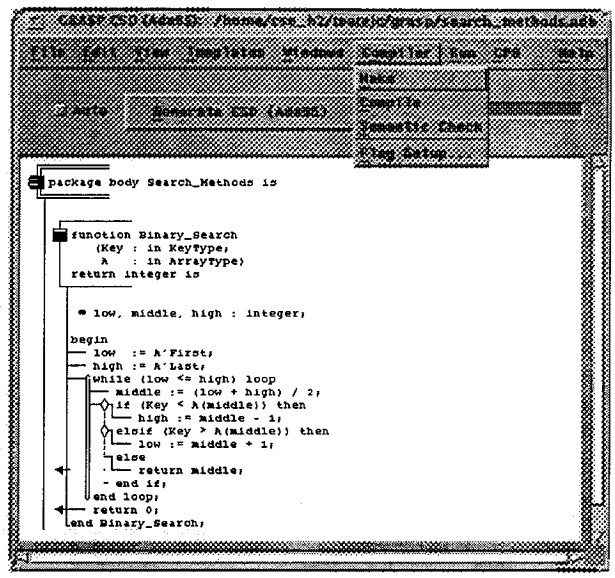

Figure 24. Compiler Options

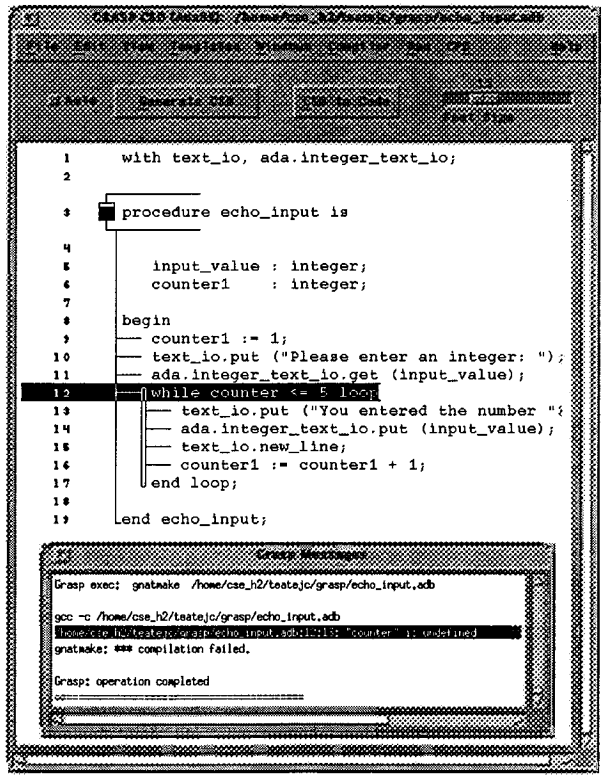

Figure 25. Error Highlighting

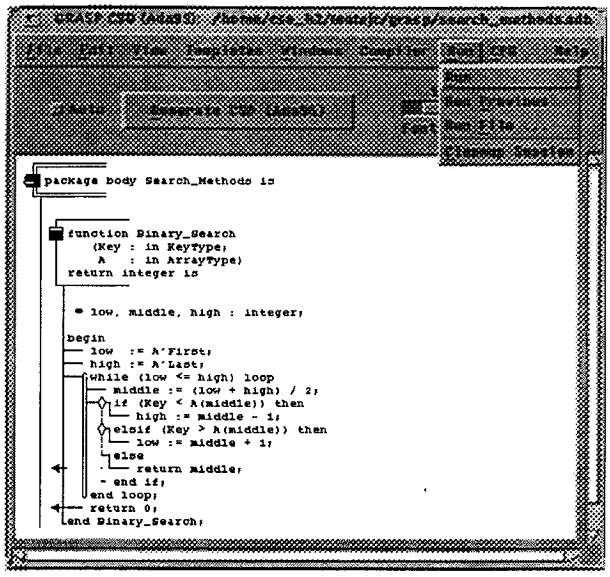

Figure 26. Run Options

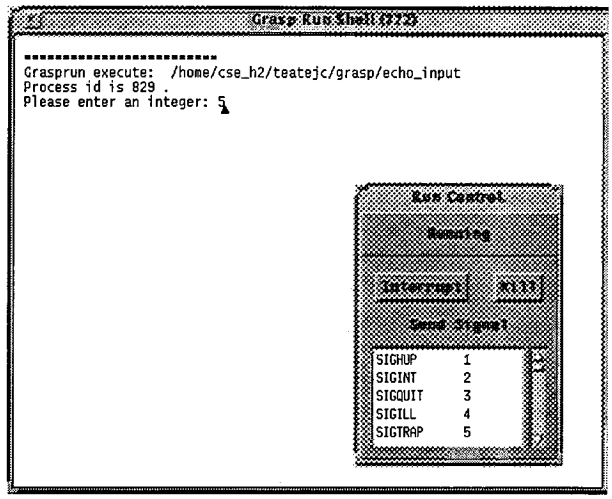

Figure 27. CSD Run / Kill

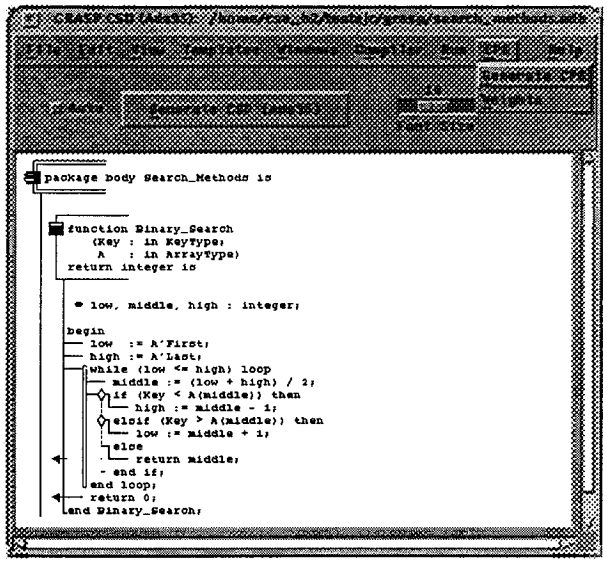

Figure 28. CPG Options

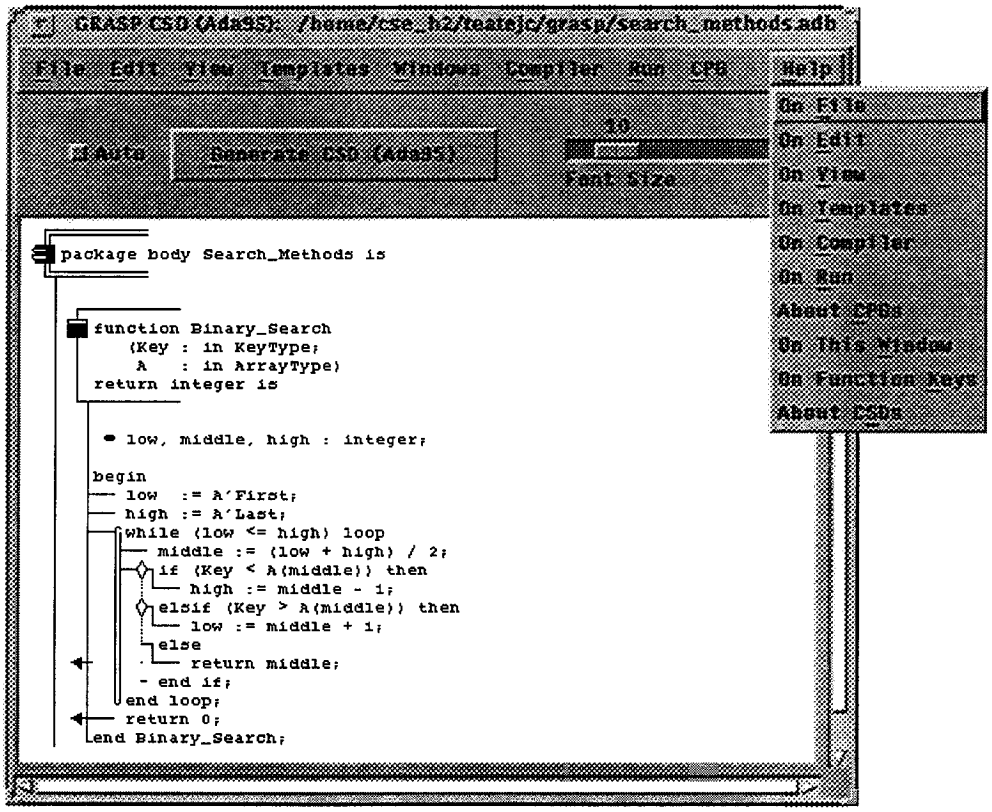

Figure 29. CSD Help Options

4. The Complexity Profile Graph

The overall goal of the GRASP project is to improve the comprehensibility of software. Thus, it is important to be able to identify complex areas of source code. The Complexity Profile Graph (CPG), a new graphical representation based on a composite of statement-level complexity metrics [McQuaid96], provides the user with the capability to quickly recognize complex areas of source code. The CPG is significant in that it shows the complexity of a program unit as a profile of statement-level complexity metrics rather than a single, global metric. For example, in the linguistic approach used in software science [Halstead77], the numbers of distinct and total operators and operands are used to compute the length and volume of a program, without regard for program structure or location within the program. The graph theoretic approach used in computing cyclomatic complexity [McCabe94] yields a metric based on the number of decisions (edges and nodes in the program graph). These traditional metrics are each single numbers used to describe an entire program unit. While there are other metrics that combine the characteristics of software science and cyclomatic complexity [Robillard89], none addresses the overall characteristics of a program unit as a visual complexity profile in the way that the CPG does.

4.1 Related Work

It is generally agreed that systematic research on metrics as tools for predicting qualitative attributes of software originated with Halstead's Software Science [Halstead77]. Halstead's basic metrics are number of unique operators, number of unique operands, total number of operators, and total number of operands. Halstead conjectures relationships between these fundamental quantities and a variety of qualitative attributes, the most popular of which are *volume* as a measure of program size, and *effort* as an indicator of psychological complexity. While widely regarded as seminal work in software complexity, Software Science fails to provide measurement at the level of detail that the CPG does.

McCabe [McCabe94] was the first to propose that "complexity depends only on the decision structure of a program" and therefore is a property derivable from a control-flow graph. The premise is that the complexity of a program, P, is related to the difficulty of performing path testing. McCabe terms this difficulty of path testing as cyclomatic complexity, v(G). A program for which v(G) exceeds a certain level was considered too big, too difficult to test, and a potential candidate for restructuring. Although widely used, the cyclomatic complexity of a program is a single number and does not provide the rich detail that the CPG does.

An attempt to provide more detail is the interconnectivity metric of [Robillard89]. The underlying model is based on the information-theory concepts of entropy and excess entropy and attempts to integrate contributions of control flow complexity, data flow complexity, and program size into a single measure. The interconnectivity metric attempts to measure the difficulty in understanding any given statement or group of statements by measuring how a given statement is related, or interconnected, to the rest of the program. The premise is that a statement is connected to the rest of the program by the variables it uses and the control structures to which it belongs.

The interconnectivity metric is computed from a matrix in which rows represent statements in the program and columns represent elements of data flow, control flow, and program size (variable definitions, variable re-definitions, and control structure). Once computed, the interconnectivity metric can be graphed as a profile of a program's statement-level complexity, much like the CPG. The CPG, however, provides greater depth and finer detail in computing complexity and, in conjunction with GRASP/Ada, provides a more flexibility and user control in exploring statement-level complexity.

The CPG is the visualization of a new and original complexity metric, based on a set of functions that describes the context, content, and scaling for complexity on a statement by statement basis. When combined graphically, the result is a composite profile of complexity for the program unit. Our current research includes the development and refinement of the associated functions, and the development of a CPG generator prototype for the GRASP/Ada software engineering tool. In this prototype, the Control Structure Diagram (CSD) [Cross96b] and the CPG are synchronized so that the visualization of control structure, nesting, and source code in the CSD window is directly linked to the corresponding visualization of statement level complexity in the CPG window. Thus, as the user scrolls through the CSD, reading and comprehending the source code, the corresponding CPG provides additional complexity information for each statement in the CSD window. Alternatively, as the user scrolls through the CPG to identify areas of high complexity, the CSD is automatically scrolled to display the corresponding source statements.

The current research provides the capability for the user to generate CSDs and CPGs from Ada 95 source code with a level of flexibility suitable for experimentation, evaluation, and practical application. For example, when the user edits the Ada source code in the CSD window, both the CSD and CPG are regenerated and rendered essentially as quickly as the user can click the generate button. It is anticipated that the CSD and CPG will provide for an increased level of comprehension and analysis during detailed design, implementation, verification, testing, and maintenance.

4.2 The Complexity Profile Graph

The CPG is based on a profile metric which is designed to compute complexity at various levels of *granularity* based on the underlying source language. We will call these various levels of granularity *measurable units* of software. The fundamental idea of the profile concept is that software can be partitioned into a set of measurable units in such a way that each token belongs to exactly one such unit. For example, an Ada 95 program is grammatically partitioned into individual program units (package, subprogram, task, etc.), and each of these can be further partitioned into statements, etc.. Theoretically, complexity can be calculated for any level of granularity defined by the grammar of the source language. In our present research, we calculate complexity at the production level in the source language grammar.

The CPG is a *visualization* of the complexity of a program unit, divided into a set of measurable units of software which we call *segments*. CPG segments consist of simple statements (e.g., assignment, procedure call) and clauses (e.g., declarations) in the underlying source language. A program unit is parsed into a set of non-overlapping segments such that each token is included in exactly one segment.

While simple statements and declarations correspond to only one segment, compound statements such as loops and selection structures are partitioned into two or more CPG segments for measurement. For example, the Ada 95 while loop shown in Figure 30 is partitioned into the four CPG segments:

1. `while (A < B) loop`
2. `Do_Something (To => B);`
3. `A := A + 1;`
4. `end loop;`

```
while (A < B) loop
    Do_Something (To => B);
    A := A + 1;
end loop;
```

Figure 30. Example Ada 95 while loop

This partitioning is advantageous for at least two reasons: these segments are the constructs which readers would generally comprehend as single units, and the natural link between the CPG and lines of source code is preserved. The general categories of CPG segments for Ada 95, except for tasking and object-oriented constructs, are shown in Table 1. Notice that the statements and constructs which are partitioned into multiple segments may have other CPG segments between their own. For example, an *if* statement could have a CPG segment for an assignment statement between its "IF condition THEN" segment and its "ELSIF condition THEN" segment.

The Profile Metric. Unlike a traditional metric which is a single function, the profile metric is actually a *family* of measurement functions,

$$F = \{ \mu_X \mid \mathbf{X \text{ is a measurable unit}} \}$$

where each member of F calculates a specific aspect of X's complexity. While the precise nature of the functions in F is still evolving, it is clear that, at a minimum, the number of tokens in the measurable unit and their inherent complexities should be considered. Further study is required to quantify the contribution of additional factors and to extend the model. For the remainder of this article we will discuss calculating the profile complexity metric for the measurable units "program unit", $\mu_{prog\_unit}(P)$, and "segment", $\mu_{segment}(S)$.

Table 1. CPG Segments for a subset of Ada 95

Statements Having Only One Segment	Statements Having Multiple Segments	Other Constructs Partitioned Into Multiple Segments
· assignment statement	**If statement**	**Record declaration**
· delay statement	· IF condition THEN	· TYPE ... IS
· exit statement	· ELSIF condition THEN	· RECORD
· goto statement	· ELSE	· END RECORD;
· null statement	· END IF;	**Generic Procedure**
· procedure call	**Case statement**	· [GENERIC]
· raise statement	· CASE expression IS	· PROCEDURE ... IS
· return statement	· WHEN choices =>	· BEGIN
· declarations	· END CASE;	· END [Name];
	Loop statements	**Generic Function**
	· [Name][Itr] LOOP	· [GENERIC]
	· END LOOP [Name];	· FUNCTION ... IS
	Block statement	· BEGIN
	· [Name][DECLARE]	· END [Name];
	· BEGIN	
	· END [Name];	

Contributions to Complexity. The profile complexity of a measurable unit is a combination of its *content* complexity and its *context* complexity. The content and the context complexities should be independent of each other. The content complexity tries to measure the amount of information within a measurable unit, e.g., token or segment; while the context

complexity tries to measure the location of a measurable unit within the source code. The complexity profile graph is designed such that the context complexity is the baseline complexity, with the content complexity riding on this baseline. The rationale of this design is to provide easy identification *clusters*, groups of contiguous segments of high complexity, which are based on the context. When a cluster is identified, the content complexity can be used to isolate the *heavy* segments in the cluster. With this design in mind, the range of magnitude of the context complexity should be larger than that of the content complexity. Currently, the content contribution is constrained to be between 0 and 3, while the context contribution is constrained to be between 0 and 15. This design provides the effect that the content complexity is a ripple riding on the curve of the context complexity.

Content Complexity $\eta(S)$. The content metric measures the quantity of information in a unit, not the quality. The measurement of content quality would require semantic analysis of the code. An example of such semantic analysis is the reference to an identifier as a variable versus a function call, or in discriminating between references to different variable identifiers based on the complexity of their underlying data types. In our present research, the content quality is assumed to be constant across all measurable units. For example, references to Car_1 of type Real_Car and Car_2 of type Toy_Car will be treated the same, i.e., having the same token content complexity. Although type Real_Car may be much more complicated than type Toy_Car, the difficulty of creating and referencing an instance of either variable is the same from the viewpoint of a programmer. Hence the content complexities of the *references* to Car_1 and Car_2 are treated as being the same, while the context complexities of the *declarations* of their types, Real_Car and Toy_Car, may be different.

Although the content complexity for most tokens is indeed held constant, i.e., 1.0, there are a few exceptions: left parenthesis, logical operators (e.g., *and*, *or*, *not*), and comparison operators (e.g., >, <, =). A left parenthesis normally indicates a compound expression, an index to an array, or a parameter for a call to a procedure, function, or entry. Thus, a left parenthesis generally adds a level of detail to be further understood, thereby increasing complexity. Since a right parenthesis always corresponds to a left parenthesis, and generally marks the end of greater detail, thereby decreasing complexity, it is treated as a regular token. Also, a logical operator combines two conditions into one (except operator *not*), so it is heuristically more error prone and complex. Comparison operators are treated similarly. Contribution weights for Ada 95 tokens are summarized in Table 2.

The content complexity, $\eta(S)$, of a CPG segment **S** is defined as the natural logarithm of the summation of all of its tokens' weight contributions.

$$\eta(S) = \ln \sum_{T \in S} \text{Weight}(T)$$

With this definition, the summation portion for most segments should be under 20 and the logarithm function will yield a value of less than 3.0.

Context Complexity $\chi(S)$. The context complexity provides the baseline level of complexity for segments of simple statements nested within a compound statement, which itself may be nested several levels deep. The context complexity of a segment will be the summation of the complexities of all the compound statements in which it resides. This means each compound statement contributes to the overall level of the complexity platform which is uniform for statements within it.

The complexity of a compound statement is based on three aspects: *inherent complexity*, *reachability*, and *breadth*. The inherent complexity, **I**, measures the difficulty and/or complexity nature of a compound statement. It is a subjective measurement. The rationale is that certain types of compound statements are more error prone than others. The inherent complexity weights in Table 3 have been used as a starting point.

The reachability complexity, **R**, indicates the difficulty of reaching a statement with respect to its path predicate. The path predicate is expressed as a set of conditions, and hence **R** is defined as the sum of the individual boolean condition complexities. The complexity of each boolean condition is calculated as the number of logical operators + 1. Although certain compound statements, e.g., ACCEPT, need an execution rendezvous to be reached, that is not considered in this complexity. Instead, it is concluded in the inherent complexity. Complexity **R** is used for the compound statements such as WHILE, IF-THEN-ELSE, and CASE-WHEN. The Breadth complexity, **B**, represents the amount of computation involved in a compound statement, and is approximated by the number of statements nested within the compound statement.

Table 2. Token Weights

Token Description	Symbol	Wt.
Logical Operators	and, or, not, etc	1.5
Comp. Operators	<, >, =, <=, etc.	1.5
Left Parenthesis	(	1.3
Identifiers	var1, proc1, etc.	1.0
Others	+, -, *, /,), var1, etc.	1.0

Delimiters and punctuation such as the comma, semicolon, colon, etc. are not included.

Table 3. Inherent Complexity Weights

Compound Statement	Wt.
SELECT, ACCEPT	4
CASE, IF, ELSIF	3
WHILE	3
FOR, basic LOOP, EXIT	2
Block	1
Others	0

These three complexities are combined in the following way for a segment S within a compound statement Y.

$$\chi(S) = c_1 * I(Y) + c_2 * R(Y) + c_3 * B(Y)$$

with weighting coefficients $c_1 = 1.0$, $c_2 = 1.0$, and $c_3 = 0.1$.

CPG Segment Profile. Combining the content complexity, $\eta(S)$, and the context complexity, $\chi(S)$, gives the profile metric, $\mu(S)$, for a segment. That is,

$$\mu(S) = s_1 * \eta(S) + s_2 * \chi(S)$$

where scaling factors, s_1 and s_2, are set to 1.0 for the examples. These scaling factors, s_1 and s_2, and the weighting coefficients from $\chi(S)$ above provide a means for adjusting the impact that individual factors have on the overall profile of the segment. To facilitate experimentation and evaluation, GRASP/Ada provides a dialog box that allows the user to manipulate the value of each scaling factor and weighting coefficient.

Program Unit Profile. Perhaps more useful than a profile at the segment level is a complexity profile at the program unit level. The complexity profile of program unit **P** is a composite of the profile metrics of its segments. The CPG is a histogram visualization of this composite. For example, consider the CSD for procedure BinarySearch in Figure 31 and the corresponding CPG in Figure 32. While the CSD shows the actual source code with the *while* loop and nested *if* statement depicted graphically, the CPG shows the complexity of the procedure as a profile of the individual statements' complexities. The recognizable complexity density in the CPG indicating the *while* loop with nested *if* statement is a cluster, as defined earlier. This visual representation of complexity (i.e., profile and cluster) forms the basis for all intended applications. Therefore, $\mu_{prog\_unit}(P)$ is referred to as the *profile metric*. Figure 33 contains the CSD and CPG of a more complex program unit, an Ada task with a rendezvous. Again, the CPG shows clusters of program complexity; however the level is much higher as one would expect with deeper nesting and the greater inherent complexity of a selective accept statement (note that the vertical scale is different).

The CPG for a program unit is displayed by plotting $\mu(S)$ values for each segment of the program unit as a histogram. In addition, the $\chi(S)$, $\eta(S)$, **I(S), R(S),** and **B(S)** graphs can be plotted separately, with or without scaling, to provide additional complexity profiles. This allows a user to view the complexity of a program from a desirable perspective. A user can choose either a color-coding or a pattern scheme to clearly distinguish the different elements of complexity when they are plotted separately. Figure 34 and Figure 35 illustrate this feature. Notice that these two figures, along with Figure 32, are simply different views of the complexity of the BinarySearch procedure in Figure 31.

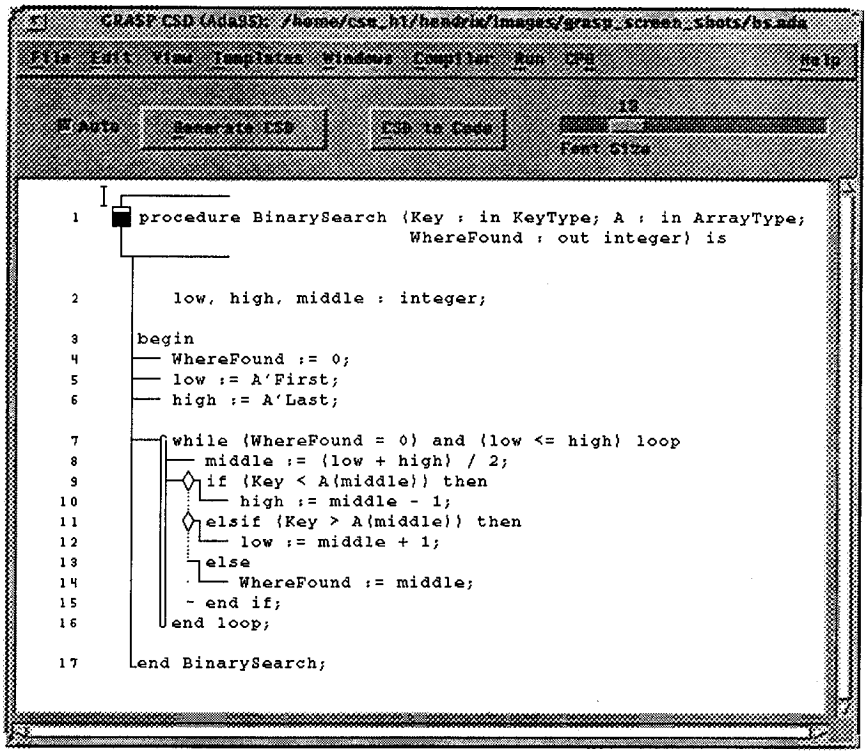

Figure 31 . BinarySearch CSD in GRASP/Ada

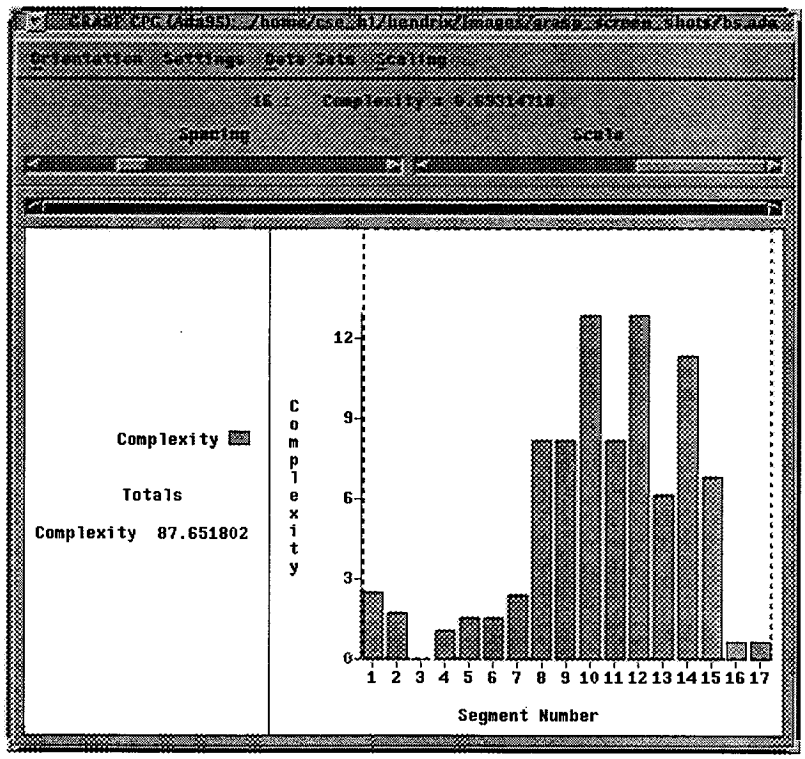

Figure 32. BinarySearch CPG in GRASP/Ada

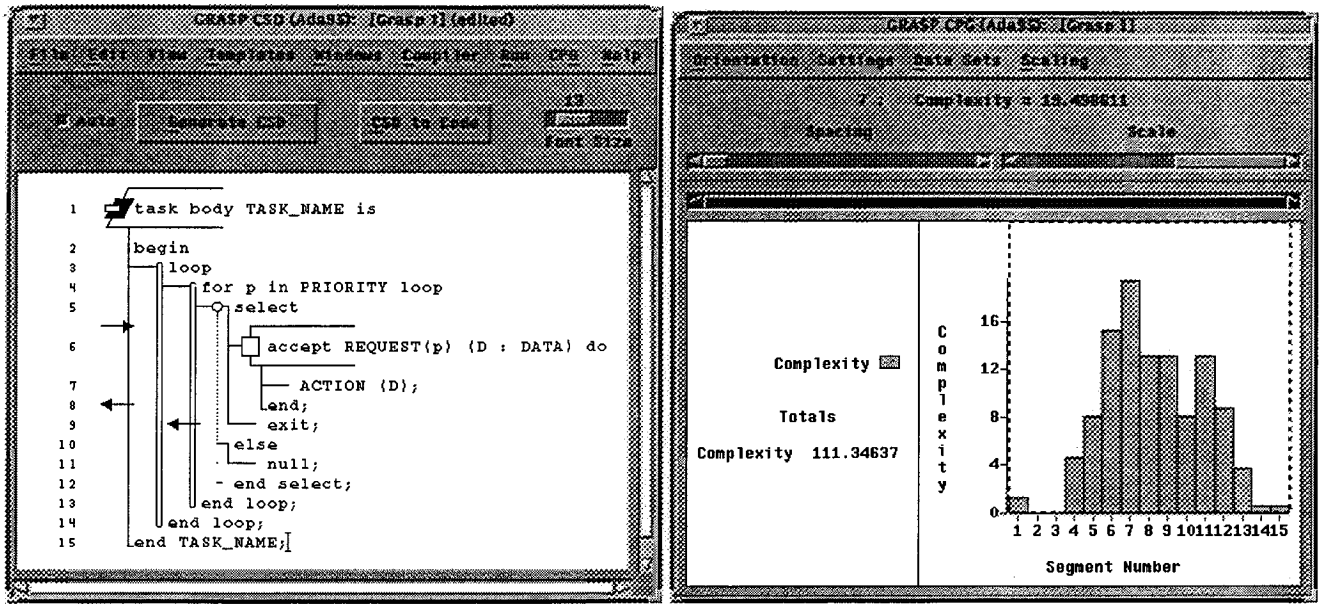

Figure 33. Synchronized CSD and CPG in GRASP/Ada

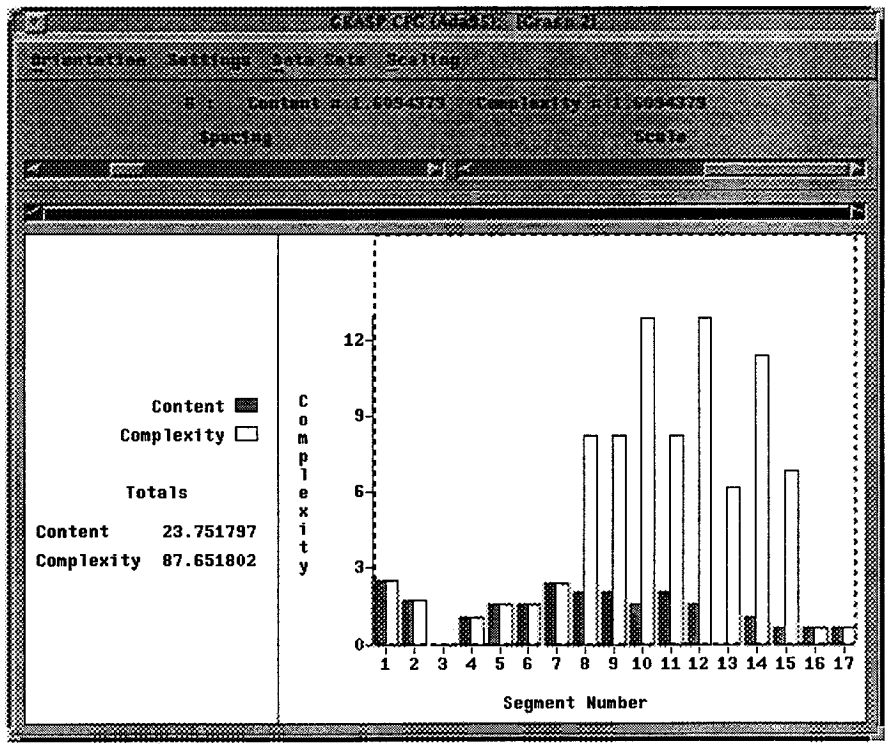

Figure 34. CPG showing Content and Total Complexity plotted separately

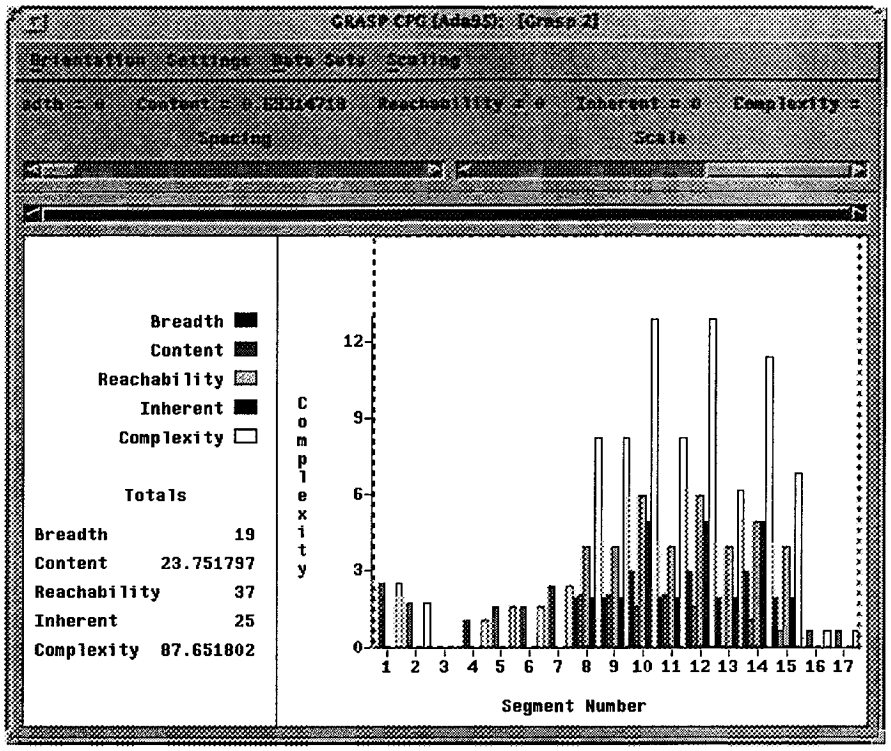

Figure 35. CPG showing all five metrics plotted separately

4.3 CPG - Future Directions

The preceding examples, while small, illustrate the potential of the CSD/CPG visualization of source code. However, we feel that the greatest advantage of this visualization lies in the reverse engineering of large software systems. The gestalt effect of the CPG visualization of a large software system would give the software engineer the ability to quickly identify complex clusters. Once these clusters were identified, the software engineer could then quickly navigate to them in the CSD window and automatically have a visual aid in comprehending the code.

Figure 36 shows the complexity profile graph of a software system with over 3700 lines of code displayed in GRASP/Ada. The CPG window clearly shows the complex clusters and the CSD window provides an automatic control flow visualization of these areas of code. Although this size program cannot be considered large, it serves nicely as an example of the visual leverage gained from the CPG. More work is needed to fully explore the issues involved with the visualization and measurement of large software systems.

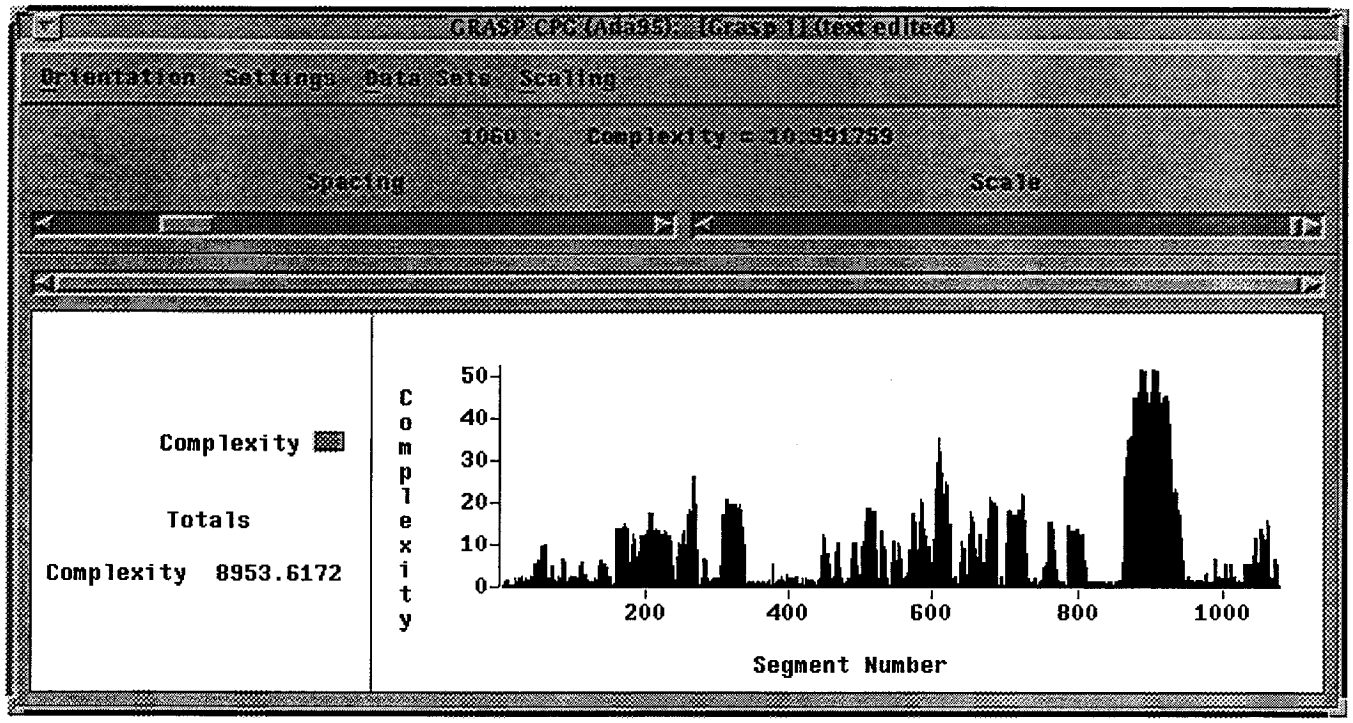

Figure 36. Complexity Profile Graph for a larger program

The CPG provides more *useful* information than traditional metrics, by incorporating both the content and context complexities into the metric. It seeks to identify not only complex statements, but also complex sets of statements (clusters). Once the clusters are identified, paths to reach this code can be identified using the CSD. The primary theme of all applications of the CPG is to locate and prioritize clusters for selective consideration where exhaustive review is impractical and to concentrate efforts on denser regions. This information has direct application to the areas of software design, implementation, testing, and maintenance, and to the software development process itself as a form of continuous feedback for analysis.

5. Tool Verification

Visualization and measurement tools such as GRASP/Ada are non-trivial to develop. There are many, often subtle, details in which the tool could produce incorrect results. Therefore it is important that tools such as GRASP/Ada be verified as being robust enough for practical application. GRASP/Ada is capable of performing a self-test in which it runs in batch mode without the graphical user interface and processes specified directories of Ada 95 source code.

During this self-test, each component of GRASP/Ada is tested, including the lexer/parser, CPG segmenter, and the CSD generator. Each file in the specified directories is parsed and all parse errors are reported. Any errors reported on code known to be correct reveal errors in GRASP/Ada's lexer/parser component. To compute the complexity profile correctly, each token in the source code must be included in exactly one segment; that is, all segments must be non-overlapping and must cover the code. Thus, in the self-test CPG segments are generated for each file and errors are reported if any segments overlap or if there is a token not included in a segment.

To test the correctness of CSD generation, the four distinct CSD views which are available to users are generated for each file. Correctness is verified both during and after generation of the diagram. Any errors reported during CSD generation indicate faults in the lexer/parser and rendering algorithm components. After generation, the diagram is checked against approximately 300 rules which state the necessary properties of vertically consecutive CSD characters in a valid diagram. Any violations of these rules plus all breaks in vertical lines of the diagram are reported. All unit symbols, which are not covered in the rulebase, are checked for correct placement in the diagram. Finally, to verify that GRASP/Ada is only adding the CSD to the source code and not altering the code from its original form, a byte-level scan of the source code in the diagram is compared to the original. Any discrepancies are reported.

As a test suite for the GRASP/Ada self-test we chose the Ada Compiler Validation Capability (ACVC) suite. The ACVC consists of positive tests (correct code) and negative tests (code with syntactic and semantic errors). The positive tests are in 2,205 files containing 293,669 lines of code and are processed by the GRASP/Ada self-test in approximately two minutes on a Sun Ultra Sparc. The negative tests are in 1,233 files containing 84,987 lines of code and are processed by the GRASP/Ada self-test in approximately 50 seconds on a Sun Ultra Sparc.

GRASP/Ada has successfully passed the self-test on all positive and negative ACVC test files. We feel that this rigorous verification process is important for all software engineering tools and we will continue to demand this level of robustness and efficiency from GRASP/Ada.

6. Summary and Future Work

The emphasis of the GRASP/Ada project is on the automatic generation of the CSD and CPG from Ada source code to support software life cycle activities. These life cycle activities should be greatly facilitated by an automatically generated set of formalized diagrams and charts to supplement the source code and other forms of documentation. Standish [Standish85] reported that program understanding represents a tremendous portion of the cost of maintenance, and Selby [Selby85] found that code reading was the most cost effective method of detecting errors during the verification process when compared to functional testing and structural testing. Code reading is still a popular and viable verification and testing strategy, as evidenced by current literature [Basili87, Ebenau94, Knight94, Seddio93, Weller93]. Hence, improved comprehension efficiency resulting from the integration of graphical notations and source code could have a significant impact on the overall cost of software production.

Version 5 (November, 1996) of the GRASP/Ada prototype provides the capability for the user to generate CSDs and CPGs from Ada source code with a level of flexibility suitable for practical application in UNIX environments. GRASP/Ada has been verified through a rigorous testing process using the Ada Compiler Validation Capability suite. A robust prototype such as GRASP/Ada is essential for the evaluation of the CSD and CPG on any non-trivial Ada 95 software.

Version 5 is currently being used as a front-end for GNAT in three to five computer science and engineering courses per quarter at Auburn University. The use of the GRASP environment in these courses is being studied to assess its overall utility. The local version has been instrumented to automatically collect usage data, which will be analyzed to determine how and how much GRASP is being used. A survey is planned in which students will be asked to indicate how they used GRASP and their preferences for its different modes. The survey data will be compared to the actual usage data. Of particular interest, will be the students' utilization of the CSD rather than plain text for displaying and printing their source code. The results of this study will be presented in a future paper.

Many software systems are not composed of programs written in one language. Rather, multiple languages are often used in the construction of large software systems. To be of practical use, a tool such as GRASP must be readily extensible to other languages and easily used in multi-lingual environments. We have developed a language independent framework for tools such as GRASP which aids in extending their functionality to multiple languages [Cross96a]. Currently we have a prototype GRASP tool that visualizes C and Java source code in addition to Ada 95. C++ and VHDL will be supported in a future release.

GRASP runs under Solaris and Linux, and is available via the internet at the Web address on the cover of this report.

References

[ACT96] "Introduction to GNAT," *Release Documents for GNAT Version 3.07*, Ada Core Technologies, 1996.

[Aoyama89] M. Aoyama, et al., "Design Specification in Japan: Tree-Structured Charts," *IEEE Software*, Mar. 1989, 31-37.

[Barnes84] Barnes, J. G. P., *Programming in Ada, Second Edition*, Menlo Park, CA: Addison-Wesley, 1984.

[Baecker90] Baecker, R. M. and Marcus, A., *Human Factors and Typography for More Readable Programs*, ACM Press, 1990.

[Barnes84] Barnes, J. G. P., *Programming in Ada, Second Edition*, Menlo Park, CA: Addison-Wesley, 1984.

[Basili87] Basili, Victor, and Selby, Richard, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, December 1987, Vol. SE-13, No.12, pp.1278-1296.

[Booch94] Booch and D. Bryan, *Software Engineering with Ada*, 3rd ed., Benjamin/Cummings, 1994.

[Cross92] J. H. Cross, E. J. Chikofsky and C. H. May, "Reverse Engineering," *Advances in Computers*, Vol. 35, 1992, 199-254.

[Cross94] J. H. Cross, "Improving Comprehensibility of Ada with Control Structure Diagrams," *Proceedings of Software Technology Conference*, April 11-14, 1994, Salt Lake City, UT (distributed on CD-ROM), 25 pages.

[Cross95] J. H. Cross and T. D. Hendrix, "Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software," *Proceedings* of Working Conference on Reverse Engineering, July 16-19, 1995, Toronto, 2-6.

[Cross96a] Cross, J. H., and Hendrix, T. D., "Language Independent Program Visualization," Eades, P. and Zhang, K. (eds.) *Software Visualization*, World Scientific Publishing Co., in press, 1996.

[Cross96b] Cross, J. H., Chang, K. H. and Hendrix, T. D. "GRASP/Ada95: Visualization with Control Structure Diagrams. *CrossTalk Defense Software Engineering Journal*, Vol. 9, No. 1, 1996, pp. 20-24.

[Ebenau94] Ebenau, Robert, "Predictive Quality Control With Software Inspections", *CrossTalk Defense Software Engineering Journal*, June 1994, pp.9-16.

[Green91] Green, T. R. G., Petre, M., and Bellamy, R. K. E., "Comprehensibility of Visual and Textual Programs," *Empirical Studies of Programmers Fourth Workshop*, Ablex, 1991.

[Green92] Green, T. R. G. and Petre, M., "When Visual Programs Are Harder to Read Than Textual Programs," *Proceedings of the Sixth European Conference on Cognitive Ergonomics (ECCE-6)*, Budapest, Hungary, 1992.

[Halstead77] Halstead, M. H., *Elements of Software Science*, Elsevier North Holland, New York, 1977.

[Knight94] Knight, J. C. and Littlewood, B., "Critical Task of Writing Dependable Software", *IEEE Software*, January 1994, Vol.11, No.1, pp.16-20.

[McCabe94] McCabe, T. and Watson, A., "Software Complexity", *CrossTalk Defense Software Engineering Journal*, December, 1994, pp.5-9.

[McQuaid96] McQuaid, P. A., *Profiling Software Complexity*, Ph.D. Dissertation, Auburn University, 1996.

[Martin85] Martin J. and McClure C., *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ : Prentice-Hall, 1985.

[Nassi73] Nassi I. and Shneiderman, B., "Flowchart Techniques for Structured Programming," *ACM SIGPLAN Notices*, Vol. 8, No. 8, August 1973, 12-26.

[Orr77] Orr, K., *Structured Systems Development*, Yourdon Press, New York, 1977.

[Petre95] Petre, M., "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM*, Vol. 38, No. 6, 1995, pp. 33-44.

[Price93] Price, B. A., Baecker, R. M., and Small, I. S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3, 1993, pp. 211-266.

[Robillard89] Robillard, P. N. and Boloix, G., "The Interconnectivity Metrics: A New Metric Showing How a Program Is Organized," *The Journal of Systems and Software*, 10, 1989, pp. 29-39.

[Scanlan89] D. A. Scanlan, "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison," *IEEE Software*, Sep. 1989, 28-36.

[Seddio93] Seddio, C., "Integrating Test Metrics Within a Software Engineering Measurement Program at Eastman Kodak Company: A Follow-Up Case Study", *Journal of Systems Software*, Vol.20, 1993, pp.227-235.

[Selby85] Selby, R. et. al., "A Comparison of Software Verification Techniques," NASA Software Engineering Laboratory Series (SEL-85-001), Goddard Space Flight Center, Greenbelt, Maryland, 1985.

[Shu88] Nan C. Shu, *Visual Programming*, New York, NY, Van Norstrand Reinhold Company, Inc., 1988.

[Standish85] Standish, T., "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, SE-10 (9), 494-497, 1985.

[Tripp89] L. L. Tripp, "A Survey of Graphical Notations for Program Design -An Update," *ACM Software Engineering Notes*, Vol. 13, No. 4, 1989, 39-44.

[Weller93] Weller, Edward F., "Lessons From Three Years of Inspection Data", *IEEE Software*, September 1993, Vol.10, No.5, pp.38-45.